1. **Class and object**

For a basic introduction, one concept is about class and object. To build a building, we need a blueprint. A class is like the blueprint, an object is like the actual building. We can of course, use the same blueprint to build many buildings.

A class starts with a upper case letter. An object starts with a lower case letter.

In defining a class, we define variables and operations (functions) on these variables. The variables are called instance variables. The operations (functions) are called methods.

For example, in the Blob class (or Node class), we define the instance variables as the x, y coordinates. The methods include draw (to draw the node on the screen), step ( to move the node, or change the coordinates), apply load, etc.

After we define the Blob class, we can declare objects which belongs to that class. Such as:

Blob blob1;

Blob blob2;

Here blob1, blob2 are objects. To use the method that we defined in the Blob class on object blob1, blob2, we use "." to call the methods. Such as:

blob1.step();

blob2.draw();

blob1.applyload();

Another example, in the Cell class, the instance variables are an array of blobs (object blob, not class). Here we use objects from Blob class as instance variables in Cell class. The methods include apply elastic force, draw the cell on screen, apply volume conservation forces, get the i-th blob, etc. For example, after we define Cell class, we can declare objects such as:

Cell cell1;

Cell cell2;

cell1.elastic();

cell2.draw();

Blob ith\_blob = cell1.getBlob(i);

Then, we have this CellGUI class which has the timer that ticks every 50 milliseconds. The timer is a built-in function and we don't need to know the details. All we need to know is that when the timer ticks, the function "handle timer" is called.

The instance variables of CellGUI class are an array of cells (object cell, not class).

The handle timer function looks like this:

**public void** handleTimer() {

*// calculate the value of time-dependent variables*

timeDependent();

*// apply all the mechanisms*

**for** (Cell cell : **cells**) {

cell.elastic(**elasticConstant**);

cell.osmosis(**osmosisConstant**);

cell.cornerstiff(**stiffnessConstant**, **nNode11**, **nNode12**, **nNode2**);

}

leonardJones();

constrictLoad();

lateralLoad();

*// apicalGlue();*

*// ectoLoad();*

*// yolkCell.osmosis(yolkOsmosisConstant);*

**yolkCell**.hydrostatic(**yolkHydroConstant**);

*// ask the nodes to move*

**for** (BlobC blob : **blobCs**) {

blob.step(**delay**/1000.0);

}

writeFile();

*// Now update the GUI.*

repaint();

}

The for-loop in the function is to loop through every object in the blob array or cell array. Some methods such as lateralLoad are defined within the CellsGUI class so that we can just call it directly without referring to some cell objects.

Of course, to initiate the whole process, we declare an object from the CellsGUI class and call the start timer function:

CellsGUI cellsGUI;

CellsGUI.startTimer();

In JAVA, every file is a class. In the CellGUI class, there is a main function which runs first. In this main function, we declare the cellGUI object and call the startTimer function.

1. **Constructor**

In the methods that we define within a class, there is a special one. This one has the same name as the name of the class. This method is the constructor and is called first when we initialize an object of this class. In a constructor, we mostly initialize the instance variables. We can also do other things in the constructor as well.

As a simple example, when we initiate an object from Blob class, we may want to define the x, y coordinates. So the constructor can be:

**public** Blob(**double** x1, **double** y1) {  
 **x** = x1;  
 **y** = y1;  
}

In this way, when we declare an object from Blob class, we call this constructor and pass two variables into the object as instance variables, which are x, y. We declare an object from Blob class like this:

Blob blob1 = **new** Blob(2,4);

Similarly, when we declare an object from the Cell class, we may want to pass in all the nodes on the membrane into this object as the instance variables.

*// constructor: this function is called first when we initiate a cell object***public** Cell(ArrayList<BlobC> blobs1) {  
 **blobs** = blobs1;  
 *// calculate the initial length of each segment, which is stored in the array "elasticLengths"* **elasticLengths** = **new** ArrayList<Double>();  
 BlobC blob1;  
 BlobC blob2 = blobs.get(0);  
 **double** x1, x2, y1, y2;  
 x2 = blob2.getX();  
 y2 = blob2.getY();  
 *// in the for loop, we first get the two adjacent blobs in the instance variable "blobs"  
 // then we calculate the initial length and put the length in the array "elasticLengths"* **for**(**int** i = 0; i < blobs.size(); i++) {  
 x1 = x2;  
 y1 = y2;  
 blob1 = blob2;  
 blob2 = blobs.get((i+1)%blobs.size());  
 x2 = blob2.getX();  
 y2 = blob2.getY();  
 **double** elasticLength = Math.*sqrt*((x1-x2)\*(x1-x2)+(y1-y2)\*(y1-y2));  
 **elasticLengths**.add(elasticLength);  
 }  
   
 *// calculate the initial area* **initialArea** = area();  
   
 *// calculate the center coordinate of the cell cx,cy* **double** sumX = 0, sumY = 0;  
 **for**(BlobC blob: blobs) {  
 sumX += blob.getX();  
 sumY += blob.getY();  
 }  
 **cx** = sumX/blobs.size();  
 **cy** = sumY/blobs.size();  
}

The constructor of Cell class does more than just initializing the instance variable “blobs”. It also calculates the initial length of each segment on the membrane. This is for the calculation of elastic forces. The constructor also calculates the initial area of the cell. This is for the calculation in the volume conservation. For convenience, we also define “elasticLengths”, “initialArea” as an instance variable. Also, this constructor calculates the center coordinates of the cell.

For CellsGUI class, we actually put the startTimer function in the constructor. We also have the meshing function in the constructor, which makes sense since we only need to initiate the mesh at the beginning. So a simplified constructor of CellsGUI looks like this:

**public** CellsGUI() {   
 *// generate mesh* meshing(); *// Timer drives the animation.* startTimer();

}

With this constructor, we can just initialize an object from CellsGUI, and the meshing method and timer will run automatically when the constructor is called:

CellsGUI cellsGUI = **new** CellsGUI();

1. **About the key word “new”**

When we initialize an object, an array, or anything, we need to allocate memory for this object/array. In Java, we use the keyword “new” to tell Java to allocate a memory.

As a little technical detail, in Java, all the primitive-type data (such as integer, floating number, string, Boolean type, etc) are stored as value. So we don’t need the “new” keyword in this case to allocate memory since the variable itself stores the value of the data.

Everything else is stored as reference (memory address). In this case, we need the “new” keyword to tell Java to allocate a new memory to store the object/array. The variable in this case only stores the memory address of the object/array, rather than the object/array itself. Sometimes when we declare a non-primitive-type variable, and we forget to “new” the variable, Java will throw us a “NullPointerException”.

One example about value vs. reference:

**public** Example(){  
 **int** i = 5; *// first we initialize variable i, i stores the value of 5* **int** j = i; *// then we initialize variable j, j stores the value of 5* i = 6; *// then we change the value of i to 6, i stores the value of 6* System.***out***.println(j); *// if we print out j, we find that j remains unchanged as 5* Blob blob1 = **new** Blob(2,4);*// first we declare a blob object, and use "new" to allocate memory  
 // now blob1 stores the address of that memory, rather than the actual memory*

Blob blob2 = blob1; *// then we declare another blob object, here we didn't use new to allocate memory  
 // here we only pass the memory address stored in blob1 to blob2  
 // so now blob2 and blob1 stores the same memory address* blob2.setX(3); *// now we change the x coordinate of blob2 to 3.   
 // since both blob1 and blob2 stores the same memory address, they actually point to the same blob object  
 // so, the x coordinate of blob1 changed as well* blob1.getX();*// this will return 3, as the x of blob1 is changed as well.* blob2 = **new** Blob(5,1); *// then, we allocate another memory for blob2, now blob2 stores the memory address of another blob object* blob2.setX(8); *// we change the x of blob2 from 5 to 8* blob1.getX(); *// this will return 3, since right now blob1 and blob2 stores different memory address, they points to different blobs now  
 // so the change of one blob will not affect the other blob* blob1 = blob2; *// now we pass the memory address stored in blob2 to blob1. So now the blob1 and blob2 points to the same memory address again  
 // the previous memory allocation of blob1 got "garbage collected" since there are no variables pointing to that memory anymore*

Blob blob3;  
 blob3.getX(); *// this will throw a null pointer exception as there is no memory address stored in blob3*

}